A parallel pattern for iterative stencil + reduce

Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Misale, C., Pezzi, G. P., & Torquati, M. (2018). A parallel pattern for iterative stencil + reduce. *The Journal of Supercomputing*, *74*(11), 5690-5705. https://doi.org/10.1007/s11227-016-1871-z

Published in:
The Journal of Supercomputing

Document Version:
Peer reviewed version

Journal of Supercomputing

A Parallel Pattern for Iterative Stencil + Reduce

--Manuscript Draft--

Manuscript Number:	
Full Title:	A Parallel Pattern for Iterative Stencil + Reduce
Article Type:	S.I. : Reengineering for Parallelism in Heterogeneous Parallel Platforms
Keywords:	parallel patterns; OpenCL; GPUs; heterogeneous multi-cores
Corresponding Author:	Maurizio Drocco University of Torino Torino, TO ITALY
Corresponding Author Secondary Information:	
Corresponding Author's Institution:	University of Torino
Corresponding Author's Secondary Institution:	
First Author:	Marco Aldinucci, Associate Professor
First Author Secondary Information:	
Order of Authors:	Marco Aldinucci, Associate Professor
	Marco Danelutto, Full professor
	Maurizio Drocco, Phd student
	Peter Kilpatrick, Professor
	Claudia Misale, Phd student
	Guilherme Peretti Pezzi, Researcher
	Massimo Torquati, Assistant Professor
Order of Authors Secondary Information:	
Funding Information:	EU FP7 REPARA project (609666) — Marco Danelutto
Abstract:	In this paper we present the Loop-of-stencil-reduce pattern as a mean for simplifying the implementation of data parallel programs on heterogeneous multi-core platforms. Loop-of-stencil-reduce is general enough to subsume map, reduce, map-reduce, stencil, stencil-reduce, and, crucially, their usage in a loop in both data parallel and streaming applications, or a combination of both. The pattern makes it possible to deploy a single stencil computation kernel on different GPUs. The paper discusses the implementation of Loop-of-stencil-reduce within the FastFlow parallel framework, considering a set of iterative data-parallel kernels running on different heterogeneous parallel systems.

Noname manuscript No.
(will be inserted by the editor)

A Parallel Pattern for Iterative Stencil + Reduce

M. Aldinucci · M.Danelutto · M. Drocco · P. Kilpatrick · C. Misale · G. Peretti Pezzi · M. Torquati

Received: date / Accepted: date

Abstract In this paper we present the *Loop-of-stencil-reduce* pattern as a mean for simplifying the implementation of data parallel programs on heterogeneous multi-core platforms. *Loop-of-stencil-reduce* is general enough to subsume *map*, *reduce*, *map-reduce*, *stencil*, *stencil-reduce*, and, crucially, their usage in a loop in both data parallel and streaming applications, or a combination of both. The pattern makes it possible to deploy a single stencil computation kernel on different GPUs. The paper discusses the implementation of *Loop-of-stencil-reduce* within the FastFlow parallel framework, considering a set of iterative data-parallel kernels running on different heterogeneous parallel systems.

Keywords parallel patterns, OpenCL, GPUs, heterogeneous multi-cores

1 Introduction

Data parallelism has played a paramount role in application design from the dawn of parallel computing. *Stencil* kernels are the class of (usually iterative) data parallel kernels which update array elements according to some

M. Danelutto and M. Torquati
Dep. of Computer Science, University of Pisa, Italy
E-mail: {marcod, torquati}@di.unipi.it

M. Aldinucci, M. Drocco and C. Misale
Dep. of Computer Science, University of Turin, Italy
E-mail: {aldinuc, drocco, misale}@di.unito.it

P. Kilpatrick
Dep. of Computer Science, Queen's University Belfast, UK
E-mail: p.kilpatrick@qub.ac.uk

G. Peretti Pezzi
Swiss National Supercomputing Centre, Switzerland
E-mail: gppezzi@gmail.com

fixed access pattern. The stencil paradigm naturally models a wide class of algorithms (e.g. convolutions, cellular automata, simulations) and it typically requires only a fixed-size and compact data exchange among processing elements, which might follow a weakly ordered execution model. The stencil paradigm does not exhibit true data dependencies within a single iteration. This ensures efficiency and scalability on a wide range of platforms ranging from GPUs to clusters. GPUs are widely perceived as data-parallel computing systems [16] so that GPU kernels are typically designed to employ the *map-reduce* parallel paradigm. The *reduce* part is typically realised as a sequence of partial GPU-side reduces, followed by a global host-side reduce. Thanks to GPUs' globally shared memory, a *map* computation can implement a stencil as a data overlay with non-empty intersection, provided they are accessed in read-only fashion to enforce deterministic behaviour. Often, this kind of kernel is iteratively called in host code in a loop body up to a convergence criterion.

Data parallelism have been provided to application programmers by way of different code artefacts (constructs, from now on) both in the shared-memory and message-passing programming models (e.g. compiler directives, skeleton frameworks, pattern libraries). Its implementation is well understood for a broad class of platforms, including GPUs (see Sec. 2). In this setting, the possibility to compose constructs certainly enhances expressivity but also the complexity of the run-time system.

We advocate composition beyond the class of data parallel constructs. We envisage parallelism exploited according to the *two tier* model [1]: *stream* and *data* parallel. Constructs in each tier can be composed and data parallel constructs can be nested within stream parallel ones. The proposed approach distinguishes itself from nesting of *task* and *data* parallelism, which has been proposed (with various degrees of integration) as a way to integrate different platforms examples include MPI+OpenMP, OmpSs+SkePU, MPI+CUDA.

In this setting, we propose the *Loop-of-stencil-reduce* pattern as an abstraction for tackling the complexity of implementing iterative data computations on heterogeneous platforms. The *Loop-of-stencil-reduce* is designed as a FastFlow [9] pattern, which can be nested in other stream parallel patterns, such as *farm* and *pipeline*, and implemented in C++ and OpenCL. We advocate *Loop-of-stencil-reduce* as a comprehensive pattern for programming GPUs in a way that is general enough to express *map*, *reduce*, *map-reduce*, *stencil*, *stencil-reduce* computations and, most of all, their usage in a loop.

The *Loop-of-stencil-reduce* simplifies GPU exploitation by taking care of a number of low-level issues, such as: device detection, device memory allocation, host-to-device (H2D) and device-to-host (D2H) memory copy and synchronisation, reduce algorithm implementation, management of persistent global memory in the device across successive iterations, and enforcing data race avoidance due to stencil data access in iterative computations. Finally, it can transparently exploit multiple GPUs on the same platform.

While this paper builds on previous results [4, 2], it advances them in several directions. The *Loop-of-stencil-reduce* pattern is an evolution of the *stencil-reduce* pattern [4]. The *Loop-of-stencil-reduce* has been refined to explicitly

include the iterative behaviour and the optimisations enabled by the awareness of the iterative computation and the possible nesting into a streaming network. Such optimizations are related to GPU persistent global memory usage, stencil and reduce pipelining, asynchronous D2H/H2D memory copies. The *Loop-of-stencil-reduce* has been uniformly implemented in OpenCL and CUDA, whereas *stencil-reduce* was dependent on CUDA-specific features not supported in OpenCL, such as Unified Memory. Also, locally-synchronous computations (by way of halo-swap) across multiple GPUs have been introduced, whereas in previous works use of multiple GPUs was possible only on independent kernel instances.

The paper itself extends [2] by introducing a complete formalisation of the *Loop-of-stencil-reduce* pattern, and a brand new experimentation plan. Specifically, the paper reports testing on three applications and three different heterogeneous platforms. Two applications exploit both stream and data parallelism. The set of platforms includes a multiple NVidia GPU Intel box and a "Big-little" Samsung mobile platform with 2 different Arm multi-core CPUs and 1 Arm GPU.

2 Related Work

Software engineers are often involved in solving recurring problems. Design patterns have been introduced to provide effective solutions to these problems. Parametric parallelism-exploitation pattern, developed to incorporate both functional and non-functional features in the pattern itself, are called algorithmic skeletons [8]. Algorithmic skeletons have been used extensively since the '90s as a means to integrate scalability in the design phase while preserving flexibility. Some of them expose task parallelism, like task-farms, pipelines and Divide&Conquer. Others, such as map, reduce and stencil, expose data parallelism. Programs using skeletons are usually much easier to optimise and to map to parallel heterogeneous architectures since the semantics of the algorithm is decoupled from the implementation.

Most skeletal frameworks (or indeed, high-level parallel programming libraries) eventually exploit either low-level tools such as NVidia CUDA or OpenCL to target hardware accelerators. CUDA is known to be more compliant to C++ and often more efficient than OpenCL. On the other hand, OpenCL is implemented by different hardware vendors such as Intel, AMD, and NVIDIA, making it highly portable and allowing the code written in OpenCL to be run on different GPUs.

Furthermore, several programming frameworks based on algorithmic skeletons have been recently extended to target heterogeneous architectures. In Muesli [11] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. StarPU [5] is focused on handling accelerators such as GPUs. Graph tasks are scheduled by its run-time support on both the CPU and various accelerators, provided the programmer has given a task implementation for each architecture.

Among related works, the SkePU programming framework is the most similar to the present work [10]. It provides programmers with GPU implementations of several data parallel skeletons (e.g. Map, Reduce, MapOverlap, MapArray) and relies on StarPU for the execution of stream parallel skeletons (pipe and farm). The FastFlow stencil operation we introduce here behaves similarly to the SkePU overlay skeleton (in some ways it was inspired by SkePU). The main difference is that the SkePU overlay skeleton relies on a SkePU-specific data type and, to the best of our knowledge, it is not specifically optimised for use inside a sequential loop. Another similar work in terms of programming multi-GPU systems is SkelCL, a high-level skeleton library built on top of OpenCL code which uses container data types to automatically optimize data movement across GPUs. In [6], the authors introduce two new SkelCL skeletons which specifically target stencil computations – MapOverlap skeleton for single-iteration stencil computations and Stencil skeleton, that provides more complex stencil patterns and possibly iterative computations. SkelCL provides data structures, called Containers, that are automatically distributed among GPUs.

In this context, the FastFlow parallel programming environment has recently been extended to support GPUs via CUDA [4] and OpenCL (as described in the present work). FastFlow CPU implementations of patterns are realised via non-blocking graphs of threads connected by way of lock-free channels [3], while the GPU implementation is realised by way of the OpenCL bindings and offloading techniques. Also, different patterns can be mapped onto different sets of cores or accelerators and so, in principle, can use the full available power of the heterogeneous platform.

Among compiler-based approaches, we recall OpenACC and OmpSs. OpenACC [15] is a compiler-based, high-level, performance portable programming model that allows programmers to create high-level host+accelerator programs without the need to explicitly initialise the accelerator, manage data transfers between the host and accelerator. It is based on compiler directives, such as pragmas, that, for instance, allow execution of a loop on a GPU by just adding the parallel loop. It also supports multi-GPU execution. The task-based OmpSS [7] extends OpenMP with directives to support asynchronous parallelism and heterogeneity, built on top of the Mercurium compiler and Nanos++ runtime system. Asynchronous parallelism is enabled by the use of data-dependencies between the different tasks of the program, and execution on multi-GPU is also supported.

3 The *Loop-of-stencil-reduce* pattern in FastFlow

In this section the semantics and the FastFlow implementation of *Loop-of-stencil-reduce* is introduced. The well-known Conway's Game-of-life is used as a simple but paradigmatic example of locally synchronous data-parallel applications (running on multiple devices).

3.1 Semantics of the *Loop-of-stencil-reduce* pattern

We assume that a is an $n-$dimensional array with sizes $d_1, \ldots, d_n$ and items of type T. We define $\alpha(f) : a = b$ where b is an $n-$dimensional array of the same size of a and items of type T' such that $\forall i_1 \in [0, d_1 - 1]; \ldots; \forall i_n \in [0, d_n - 1]$ $b_{i_1,\ldots,i_n} = f(a_{i_1,\ldots,i_n})$ and $/(\oplus) : a = v$ where

$$v = \bigoplus\nolimits_{\forall i_1 \in [0,d_1-1];\ldots;\forall i_n \in [0,d_n-1]} (a_{i_1,\ldots,i_n})$$

with f of type $T \to T'$ and $\oplus$ of type $T \times T \to T$, associative. Then we define the generic $n-$dimensional *stencil operator* as follows:

$$\sigma_k^n : a_{i_1,\ldots,i_n} = \{a'_{j_1,\ldots,j_n} \mid \forall l \; j_l \in [i_l - k, i_l + k]\}$$

$\sigma_k^n : a_{i_1,\ldots,i_n} \in T^{(2k+1)^n}$ and

$$a'_{i_1,\ldots,i_n} = \begin{cases} a_{i_1,\ldots,i_n} & \textit{iff} \quad \forall j \; i_j \in [0, d_j - 1] \\ \bot & \textit{otherwise} \end{cases}$$

With these definitions, we proceed to characterise the stencil parallel pattern functional semantics as follows[1]:

$$\textbf{stencil}(\sigma_k, f) : a = \alpha(f) \circ \alpha(\sigma_k) : a$$

possibly computing in parallel all the $\sigma_k(a_{i_1,\ldots,i_n})$ and $f(\sigma_k(a_{i_1,\ldots,i_n}))$. We remark that, in this formulation, f takes as input a neighbourhood of type $T^{(2k+1)^n}$. Moreover, both f and $\oplus$ should take into account the possibility that some of the input arguments are $\bot$. At this point we may formally define the *Loop-of-stencil-reduce* parallel pattern's functional semantics as follows:

```
1: procedure LOOP-OF-STENCIL-REDUCE((k, f, ⊕, c, a))
2:     repeat
3:         a = stencil(σ_k, f) : a
4:     until c(/⊕ : a)
5: end procedure
```

We consider this as the simplest pattern modelling iterative stencil+reduce parallel computations. Small variants of this pattern are worth consideration, however, to take into account slightly different computations with similar parallel behaviour. The first variant considered is that where the function applied in the $\alpha(f)$ phase takes as an input the "index" of the element considered (the centroid of the neighbourhood) in addition to all the items belonging to the neighbourhood. We call this variant *Loop-of-stencil-reduce-i* and it can be simply defined by the same algorithm as that of the *Loop-of-stencil-reduce* with minor changes to the auxiliary functions f and $\oplus$:

- we consider a new function $\overline{f}$ with type $T^{2k+1} \times N^{2k+1} \to T'$, and
- a new stencil operator $\overline{\sigma}_k^n$ computing

$$\overline{\sigma}_k^n : a_{i_1,\ldots,i_n} = \{\langle a'_{j_1,\ldots,j_n}, \langle j_1, \ldots, j_n \rangle \rangle \mid \forall l \; j_l \in [i_l - k, i_l + k]\}$$

[1] We omit the dimension n in σ_k^n here, as we assume the dimension n is the same as that of the array a: a single dimensional array will have $n = 1$, a 2D matrix $n = 2$, and so on.

With such definitions the *loop-of-stencil-reduce-i* is just a *Loop-of-stencil-reduce* with different parameters, that is LOOP-OF-STENCIL-REDUCE$(k, \overline{f}, \overline{\oplus}, c, a)$. The second variant of the *Loop-of-stencil-reduce* pattern we introduce changes slightly the way in which the termination condition is computed and used, to deal with those iterative computations where *convergence* of the reduced values is of interest, rather than their absolute values. Again, this can be modelled by the *Loop-of-stencil-reduce* pattern with suitable modification of the component functions. We assume that:

- f returns both the original parameter and the new computed value, i.e. if formerly we had $f : a_{i_1,\ldots,i_n} = b_{i_1,\ldots,i_n}$ we will now have instead $f' : a_{i_1,\ldots,i_n} = \langle b_{i_1,\ldots,i_n}, a_{i_1,\ldots,i_n} \rangle$
- the $\oplus$ associative function is substituted by two functions:
  - δ of type $T \times T \to T$, that is applied over all the items resulting from the $\alpha(f) \circ \alpha(\sigma_k)$ step; and
  - $\oplus$ of type $T \times T \to T$, that is used to reduce the items computed by δ to a single value to be passed to termination condition c.

With these definitions, we may define the second *Loop-of-stencil-reduce* variant as follows:

1: **procedure** LOOP-OF-STENCIL-REDUCE-D$((k, f, \delta, \oplus, c, a))$
2: **repeat**
3: $b =$ **stencil**(σ_k, f'):a
4: $d = \alpha(\delta) : b \quad a = \alpha(fst) : b$ ▷ being $fst : \langle a, b \rangle = a$
5: **until** c$(/\oplus : d)$
6: **end procedure**

It is clear that the LOOP-OF-STENCIL-REDUCE-D may be easily extended to a LOOP-OF-STENCIL-REDUCE-D-I where the $\overline{f}$ and $\overline{\sigma}_k$ functions are used in place of f and σ_k as we did to turn the *Loop-of-stencil-reduce* into *Loop-of-stencil-reduce-i*. The third and last variant we present simply consists in considering some kind of global "state" variable (such as the number of iterations) as a parameter of the termination condition:

1: **procedure** LOOP-OF-STENCIL-REDUCE-S$((k, f, \oplus, c, a))$
2: $s =$ init$(\ldots)$;
3: **repeat**
4: $a =$ **stencil**$(\sigma_k, f) : a; \quad s =$ update$(\ldots)$;
5: **until** c$(/\oplus : a$,s$)$
6: **end procedure**

and again it may be included in both the $-D$ and $-I$ versions of the *Loop-of-stencil-reduce* pattern.

With a similar methodology, we may define the functional semantics of more classical data parallel patterns such as **map** and **reduce** as follows:

- The **map** pattern computes **map**$(f) : a = \alpha(f) : a$ possibly carrying all the $f(a_{i_1,\ldots,i_n})$ computations in parallel.
- The **reduce** pattern computes **reduce**$(g) : a = /(g) : a$ possibly computing the different applications of g at the same level of the resulting reduction tree in parallel.

We remark that, from a functional perspective, **map** and **stencil** patterns are very similar, the only difference being the fact that the stencil elemental function f takes as input a set of atomic elements rather than a single atomic element. Nevertheless, from a computational perspective the difference is substantial, since the semantics of the map leads to *in-place* implementation, which is in general impossible for stencil. These parallel paradigms have been proposed as patterns for both multi-core and distributed platforms, GPUs, and heterogeneous platforms [14,10]. They are well-known examples of data-parallel patterns, since as we stated above the elemental function of a map/stencil can be applied to each input element independently of the others, and also applications of the combinator to different pairs in the reduction tree of a reduce can be done independently, thus naturally inducing a parallel implementation. Finally, we remark the basic building block of *Loop-of-stencil-reduce* (the *repeat* block at lines 3–4 of the LOOP-OF-STENCIL-REDUCE pattern above) is *de-facto* the stencil-reduce pattern previously presented in [4].

3.2 The FastFlow *Loop-of-stencil-reduce* API

In FastFlow, the *Loop-of-stencil-reduce* pattern implements the semantics described in 3.1. In particular, it implements an instance of the semantics in which the stencil-reduce computation is iteratively applied, using the output of the stencil at the i-th iteration as the input of the $(i+1)$-th stencil-reduce iteration. Moreover, it uses the output of the reduce computation at the i-th iteration, together with the iteration number, as input of the *iteration condition*, which decides whether to proceed to iteration $i+1$ or stop the computation.

The FastFlow implementation is aimed at supporting iterative data-parallel computations both on CPU-only and CPU+GPU platforms. For CPU-only platforms, the implementation is written in C++ and exploits the FastFlow map pattern. On the other hand, when an instance of the *Loop-of-stencil-reduce* pattern is deployed onto a GPU or another accelerator, the implementation relies on the OpenCL framework features. The FastFlow framework provides the user with constructors for building *Loop-of-stencil-reduce* instances, i.e. a combination of parametrisable building blocks:

- the OpenCL code of the elemental function of the stencil;
- the C++ and OpenCL codes of the combinator function;
- the C++ code of the iteration condition.

The language for the *kernel* codes implementing the elemental function and the combinator – which constitute the business code of the application – can be device-specific or coded in a suitably specified C++ subset (e.g. REPARA C++ open specification [12]). Functions are provided that take as input the business code of a kernel function (elemental function or combinator) and translate it into a fully defined OpenCL kernel, which will be offloaded to target accelerator devices by the FastFlow runtime. Note that, from our

definition of elemental function (Sec. 3.1), it follows that the *Loop-of-stencil-reduce* programming model is data-oriented rather than thread-oriented, since indexes refer to the input elements rather than the work-items (i.e. threads) space, which is in turn the native programming model in OpenCL.

In order to build a *Loop-of-stencil-reduce* instance, the user also has to specify two additional parameters controlling parallelism: 1) the number of accelerator devices to be used (e.g. number of GPUs in a multi-GPU platform) and, 2) the maximum size of the neighbourhood accessed by the elemental function when called on each element of the input. Note that the second parameter could be determined by a static analysis on the kernel code in most cases of interest, i.e. ones exhibiting a static stencil (e.g. Game of Life [13]) or dynamic stencil with reasonable static bounds (e.g. Adaptive Median Filter, [4]). Once built, a *Loop-of-stencil-reduce* instance can process *tasks* by applying the iterative computation described in Sec. 3.1 to the input of the task, by way of the user-defined building blocks. An instance can run either in *one-shot* (i.e. single task) or *streaming* (i.e. multi-task) mode. In streaming mode, independent tasks can be offloaded to different GPUs, thus exploiting inter-task parallelism. Moreover, intra-task parallelism can be employed by offloading a single task to a *Loop-of-stencil-reduce* instance deployed onto different GPUs. Although this poses some challenges at the FastFlow implementation level (see Sec. 3.3), at the API level it requires almost negligible refactoring of user code. That is, when defining the OpenCL code of the elemental function, the user is provided with local indexes over the index space of the device-local sub-input – to be used when accessing the input – along with global indexes over the index space of the whole input – to be used to e.g. check the absolute position with respect to input size.

Figure 1 illustrates a Game of Life implementation on top of the *Loop-of-stencil-reduce* API in FastFlow. Source-to-source functions are used to generate OpenCL kernels for both stencil elemental function (lines 1–12) and reduce combinator (lines 14–15). The source codes are wrapped into fully defined, efficient OpenCL kernels. The user, in order to enable exploitation of intra-task parallelism, has to use local indexes `i_` and `j_` to access elements of the input matrix. C++ codes for iteration condition and reduce combinator are not reported, as they are trivial single-line C++ lambdas. The constructor (lines 17–20) builds a *Loop-of-stencil-reduce* instance by taking the user-parameterised building blocks as input, plus the identity element for the reduce combinator (0 for the sum) and the parameters for controlling intra-task parallel behaviour, namely the number of devices to be used over a single-task (NACC) and the 2D maximum sizes of the neighbourhood accessed by the elemental function (Game of Life is based on 3-by-3 neighbourhoods). Finally, the constructor is parameterised with a template type golTask which serves as an interface for basic input-output between the application code and the *Loop-of-stencil-reduce* instance.

FastFlow does not provide any automatic facility to convert C++ code into OpenCL code. It does, however, facilitate this task via a number of features including:

```
std::string stencilf = ff_stencilKernel2D_OCL(
                    "unsigned char", "in", //element type and input
                    "N", "M",              //rows and columns
                    "i", "j", "i_", "j_",  //row-column global and local indexes
                    std::string("") +
          /* begin of the OpenCL kernel code */
                    "unsigned char n_alive = 0;\n" +
                    "n_alive += i>0 && j>0 ? in[i_-1][j_-1] : 0;\n" +
                    ... +
                    "n_alive += i<N-1 && j<M-1 ? in[i_+1][j_+1] : 0;\n" +
                    "return (n_alive == 3 || (in[i_][j_] && n_alive == 2));"
          /* end OpenCL code */);

std::string reducef = ff_reduceKernel_OCL(
                    "unsigned char", "x", "y", "return x + y;");

ff::ff_stencilReduceLoop2DOCL<golTask> golSRL(
                     stencilf, reducef, 0, iterf, // building blocks
                    N, N, NACC,  // matrix size and no. of accelerators
                    3, 3);       // halo size on the 2 dimensions
```

Fig. 1: Implementation of Game of Life [13] on top of the *Loop-of-stencil-reduce* API in FastFlow.

```
while (cond) {
  before (...)  // [H] initialisation , possibly in parallel on CPU cores
  prepare (...) // [H+D] swap I/O buffers, set kernel args, D2D-sync overlays
  stencil<SUM_kernel,MF_kernel> (input, env) // [D] stencil and partial reduce
  reduce op data // [H] final reduction
  after (...)   // [H] iteration finalisation , possibly in parallel on CPU cores
}
read(output) //[H+D] D2H-copy output
```

Fig. 2: *Loop-of-stencil-reduce* pattern general schema.

- Integration of the same pattern-based parallel programming model for both CPUs and GPUs. Parallel activities running on CPUs can be either coded in C++ or OpenCL.
- Setup of the OpenCL environment.
- Simplified data feeding to both software accelerators and hardware accelerators (with asynchronous H2D and D2H data movements).
- Orchestration of parallel activities and synchronisations within kernel code (e.g. reduce tree), synchronisations among kernels (e.g. stencil and reduce in a loop), management of data copies (e.g. halo-swap buffers management).
- Transparent usage of multiple GPUs on the same box (sharing the host memory).

3.3 The FastFlow implementation

The iterative nature of the *Loop-of-stencil-reduce* computation presents challenges for the management of the GPU's global memory across multiple iterations, i.e. across different kernel invocations.

The general schema of the *Loop-of-stencil-reduce* pattern is described in Fig. 2. Its runtime is tailored to efficient loop-fashion execution. When a task is submitted to be executed by the devices onto which the pattern is deployed, the runtime takes care of allocating on-device global memory buffers and filling them with input data via H2D copies. The naïve approach for supporting iterative computations on a hardware accelerator device equipped with some global memory (e.g. GPU) would consist in putting a global synchronisation barrier after each iteration of the stencil, reading the result of the stencil back from the device buffer (full size D2H copy), copying back the output to the device input buffer (full size H2D copy) and proceeding to the next iteration. FastFlow in turn employs *device memory persistence* on the GPU across multiple kernel invocations, by just swapping on-device buffers. In the case of multi-device intra-task parallelism (Sec. 3.2), small device-to-device copies are required after each iteration, in order to keep halo borders aligned, since no device-to-device copy mechanism is available (as of OpenCL 2.0 specification, device-to-device transfers). Global memory persistence is quite common in iterative applications because it drastically reduces the need for H2D and D2H copies, which can severely limit the performance. This also motivates the explicit inclusion of the iterative behaviour in the *Loop-of-stencil-reduce* pattern design which is one of the differences with respect to solutions adopted in other frameworks, such as SkePU [10].

As a further optimisation, FastFlow exploits OpenCL events to keep *Loop-of-stencil-reduce* computation as asynchronous as possible. No dependencies exist between stencil and reduce computations at different iterations. Put another way, stencil and reduce computations can be pipelined (i.e. stencil at iteration $i+1$ can run in parallel with reduce at iteration i). Moreover, in the case of multi-GPU intra-task parallelism, sub-tasks running on different GPUs at the same iteration are independent of each other, and so can run in parallel. By exploiting the OpenCL events API, an almost arbitrary graph of task dependencies can be implemented, thus fully exploiting all the available parallelism among operations composing a *Loop-of-stencil-reduce* computation.

4 Experiments

Here we present an assessment of the *Loop-of-stencil-reduce* FastFlow implementation. For this aim, three applications are considered: the Helmholtz equation solver based on iterative Jacobi method (Sec. 4.1), the Sobel edge detector over image streams (Sec 4.2) and the two-phase video stream restoration algorithm [4] (Sec. 4.3).

Platform	Rows	CPU	1xGPU	2xGPUs
2 eight-core Xeon @2.2GHz, 2 Tesla M2090 GPUs	512	0.31	0.31	0.32
	4096	16.99	10.84	5.88
	16384	252.67	171.84	91.46
1 eight-core Xeon @2.6GHz, Tesla K40 GPU	512	0.26	0.26	-
	4096	25.00	7.42	-
	16384	384.16	116.37	-
Cortex-A15 @2.0GHz + Cortex-A7 @1.4 GHz, Arm Mali-T628 GPU	512	3.51	6.91	-
	2048	13.87	23.83	-
	4096	64.61	92.51	-

Table 1: Execution time of the Helmholtz equation solver.

Each experiment was conducted on three different platforms: 1) an Intel workstation with 2 eight-core (2-way hyper-threading) Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, and 64 GBytes of main memory, equipped with two NVidia Tesla M2090 GPUs; 2) an Intel workstation with one eight-core (2-way hyper-threading) Xeon E5-2650 @2.6GHz, 20MB L3 shared cache, 64 GBytes of main memory, equipped with a high-end NVidia Tesla K40 GPU; 3) a small Samsung workstation with a eight-core Exynos-5422 CPU (Cortex-A15 @2.0GHz plus Cortex-A7 @1.4 GHz) equipped with a Arm Mali-T628 GPU. All systems run Linux x86_64.

The general methodology we adopt is to derive a *Loop-of-stencil-reduce* formulation of the considered problem, translate it into a FastFlow network and compare different deployments of the *Loop-of-stencil-reduce* node. Namely, we consider CPU, single-GPU and multi-GPU deployments. We remark, as we discussed in Sec. 3.3, that the CPU node is a native multi-core implementation, thus not relying on OpenCL as parallel runtime. Moreover, GPU deployments are compared to the best-case scenarios from the CPU world, thus considering the parallel configuration (e.g. thread allocation) of the FastFlow network yielding best performance. Reported execution times are in seconds.

4.1 The Helmholtz equation solver

The first application we consider is an iterative solver for the Helmholtz partial differential equation, which is applied in the study of several physical problems. The solver is a paradigmatic case of iterative 2D-stencil computation, in which each point of a read-only matrix (i.e. the input matrix) is combined with the respective 3-by-3 neighbourhood of the partial solution matrix in order to compute a new partial solution. The termination is based on a convergence criterion, evaluated as a function of the difference between two partial solutions at successive iterations, compared against a global threshold.

The implemented FastFlow network is a single *Loop-of-stencil-reduce* node executing the procedure in one-shot fashion on different input matrices. Table 1 shows the observed results. The general behaviour is an immediate

improvement resulting from the GPU exploitation. A cross-platform exception is the small matrix case, on which the same execution times are observed on CPU and GPU deployments. This is easily explained by Amdahl's law, since H2D/D2H copies to actual computation results in a non negligible ratio. Speedups carried by the K40 and the M2090 GPUs mirror the different computational capabilities of the two devices from one side and the different parallelism available on the respective platforms from the other. Moreover, on the first platform execution times on the two-GPU deployment scales almost linearly with respect to the one-GPU deployment, showing that the runtime does not induce any substantial overhead while managing data distribution on multiple GPUs. The third platform in this case shows its inefficiency for scientific applications. Since it is designed to be more suitable for low-energy multimedia architectures such as smartphones, performance decrease when more computational demand and floating point operations are needed.

4.2 The streaming Sobel edge detector

The second application we consider consists in basic and streaming variants of a classical image processing filter, namely the Sobel edge detector. The basic version is a simple convolution-like operator, which applies a 2D-stencil to each (neighbourhood of the) pixel of the input image to produce a new image, in which pixel values represent the likelihood for the pixel of belonging to an edge in the original image. As with all the convolution-like image processing filters, the Sobel detector is a paradigmatic case of non-iterative 2D-stencil computation. The streaming variant applies the Sobel filter to a series of independent images, each read from a different file.

We implemented a *Loop-of-stencil-reduce* version of the Sobel filter, which arises directly from its definition. We executed the filter in one-shot fashion to three different square input images, with different sizes. Moreover, we included a streaming version in order to both consider a more common use case and show the approach of integrating a data-parallel node (the basic Sobel filter) into a FastFlow network. The resulting FastFlow term is: pipe(read, sobel, write), where sobel is a *Loop-of-stencil-reduce* node. We run the streaming version on streams of 100 images, each built as random permutation of the input set mentioned. The different deployments have been compared over the same stream, kept constant by fixing the random seed. Because of the reduced amount of GPU memory available on the third platform, we excluded the largest image from tests.

Table 2 shows the observed results. We remark the single-iteration pattern represents the worst-case scenario for GPU exploitation, since little computation is available to hide the latency of H2D/D2H memory copies. Indeed, the CPU deployment on the first platform performs better than single-GPU one, while two-GPU deployment still yields some improvement. Conversely, the K40 GPU on the second platform is still able to improve the execution time by an average of about 3× with respect to the CPU deployment. Finally,

Platform	Width (px)	CPU	1xGPU	2xGPUs
2 eight-core Xeon @2.2GHz, 2 Tesla M2090 GPUs	512	0.33 ms	0.79 ms	1.33 ms
	4096	0.02	0.02	0.01
	16384	0.22	0.31	0.20
	Stream	11.96	16.27	11.09
1 eight-core Xeon @2.6GHz, Tesla K40 GPU	512	0.58 ms	0.68 ms	-
	4096	0.03	0.01	-
	16384	0.53	0.17	-
	Stream	27.89	8.97	-
Cortex-A15 @2.0GHz + Cortex-A7 @1.4 GHz, Arm Mali-T628 GPU	512	4.91 ms	7.02 ms	-
	4096	0.29	0.27	-
	Stream	28.22	23.45	-

Table 2: Execution time of the Sobel filter on different platforms. For each platform, the upper rows refer to the basic filter on different sizes of the input image; the last row refers to the streaming variant on 100 random images.

small improvement is carried by the Mali GPU on the third platform, while a more neat improvement is observable in the streaming variant, since in the latter case the GPU-side allocation overhead is mitigated.

4.3 The two-phase video restoration algorithm

The third and most complex application is a two-phase parallel video restoration filter. For each video frame, in the first step (i.e. the detection phase) a traditional adaptive median filter is employed for detecting noisy pixels, while in the second step (i.e. the restoration phase) a regularisation procedure is iterated until the noisy pixels are replaced with values able to preserve image edges and details. The restoration phase is based on a 2D-stencil regularisation procedure, which replaces each pixel with the value minimising a function of the pixel neighbourhood. The termination is decided on a simple convergence criterion, based on the average absolute difference between two partial solutions at successive iterations, compared against a global threshold.

We implemented the application by modelling it with the FastFlow term: pipe(read, detect, ofarm(restore), write), where ofarm is a First-In-First-Out farm and restore is the *Loop-of-stencil-reduce* implementation of the restoration procedure. Samples of 100 frames at VGA (640 × 480), 720p (1280 × 720) and HDTV (2048 × 1080) resolutions are considered as input streams and artificial noise is added to each stream, at 30% and 70% level. In order to include an example of different integrations of a *Loop-of-stencil-reduce* node into a FastFlow network, two different multi-GPU exploitation schemas are compared on the first platform, exploiting parallelism both *between* and *inside* stream items. The first case – referred as 2xGPU (b) in Table 3 – amounts at instantiating a two-worker ofarm of restoration nodes, while the latter one is the already considered two-GPU deployment of a single node.

Platform	Video	CPU	1xGPU	2xGPUs	2xGPUs (b)
2 eight-core Xeon @2.2GHz, 2 Tesla M2090 GPUs	VGA, 30%	23.74	8.69	4.59	4.64
	VGA, 70%	49.65	8.70	4.61	4.69
	720p, 30%	67.78	25.23	13.12	13.16
	720p, 70%	147.69	25.28	13.50	13.55
	1080p, 30%	162.27	60.01	30.78	30.81
	1080p, 70%	354.18	60.11	32.39	32.44
1 eight-core Xeon @2.6GHz, Tesla K40 GPU	VGA, 30%	41.56	3.41	-	-
	VGA, 70%	87.32	4.39	-	-
	720p, 30%	118.99	9.72	-	-
	720p, 70%	259.54	12.71	-	-
	1080p, 30%	285.34	23.89	-	-
	1080p, 70%	623.20	29.99	-	-
Cortex-A15 @2.0GHz + Cortex-A7 @1.4 GHz, Arm Mali-T628 GPU	VGA, 30%	373.63	144.57	-	-
	VGA, 70%	739.92	206.26	-	-
	720p, 30%	986.55	409.77	-	-
	720p, 70%	2125.89	601.42	-	-
	1080p, 30%	2730.52	974.87	-	-
	1080p, 70%	4644.86	1364.74	-	-

Table 3: Execution time of the video restoration filter on different platforms, over 100-frame samples, under different noise conditions. Different resolutions are considered, ranging from VGA to HDTV. On the first platform, the inter-frame (b) multi-GPU deployment is also considered.

Table 3 shows the observed results. As expected, the multi-iteration streaming nature exhibited by this application is profitably captured by the *Loop-of-stencil-reduce* pattern, yielding good performance on all considered scenarios. In particular, execution times on the K40 GPU on the second platform show speedups ranging from 12× to 20× with respect to the CPU deployment, delivering a throughput of about 30 frames per second for the low-noise case on VGA resolution. Analogous performance are obtained from the two-GPU deployment on the second platform, while a minimal degradation is introduced by switching to the inter-frame version, due to the slightly higher amount of synchronisations induced. The third platform provides considerable speedup in this case, confirming it is well suited to target media-oriented applications, which do not feature high numerical demand.

5 Conclusions

In this work we have presented the *Loop-of-stencil-reduce*, a parallel pattern specifically targeting iterative data-parallel computations on heterogeneous multi-cores. We first provided motivation for implementing a new pattern and then we gave a clear and rigorous semantics of the pattern. Furthermore, we showed that different iterative kernels can be easily and effectively parallelized by using the *Loop-of-stencil-reduce* on the available GPUs exploiting the OpenCL capabilities of the FastFlow parallel framework.

As a future extension of this work, we plan to build on top of the current implementation of the *Loop-of-stencil-reduce* a domain specific language (DSL) specifically targeting data parallel computations in a streaming work-flow.

Acknowledgment

This work has been supported by the EU FP7 REPARA project (no. 609666) and by the NVidia GPU Research Center at University of Torino.

References

1. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a research framework for high-performance grid programming environments. In: Grid Computing: Software environments and Tools, chap. 10, pp. 230–256. Springer (2006)
2. Aldinucci, M., Danelutto, M., Drocco, M., Kilpatrick, P., Peretti Pezzi, G., Torquati, M.: The loop-of-stencil-reduce paradigm. In: Proc. of Intl. Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms. IEEE, Helsinki, Finland (2015)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Proc. of 17th Intl. Euro-Par 2011 Parallel Processing, *LNCS*, vol. 6853, pp. 170–181. Springer, Bordeaux, France (2011)
4. Aldinucci, M., Peretti Pezzi, G., Drocco, M., Spampinato, C., Torquati, M.: Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. International Journal of High Performance Computing Application (2015)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
6. Breuer, S., Steuwer, M., Gorlatch, S.: Extend ing theSkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations, pp. 15–21. Vienna, Austria (2014)
7. Bueno-Hedo, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguadé, E., Labarta, J.: Productive programming of gpu clusters with ompss. 26th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS 2012) pp. 557–568 (2012)
8. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389–406 (2004)
9. Danelutto, M., Torquati, M.: Structured parallel programming with "core" fastflow. In: Central European Functional Programming School, *LNCS*, vol. 8606, pp. 29–75. Springer (2015)
10. Enmyren, J., Kessler, C.W.: SkePU: a multi-backend skeleton programming library for multi-gpu systems. In: Proc. of the fourth Intl. workshop on High-level parallel programming and applications, HLPP '10, pp. 5–14. ACM, New York, NY, USA (2010)
11. Ernsting, S., Kuchen, H.: Data parallel skeletons for gpu clusters and multi-gpu systems. In: Proc. of PARCO 2011. IOS Press (2011)
12. Garcia, J.D.: REPARA C++ open specification. Tech. Rep. ICT-609666-D2.1, REPARA EU FP7 project (2-14)
13. Gardner, M.: Mathematical games: the fantastic combinations of John Conway's new solitaire game 'Life'. Scientific American **223**(4), 120–123 (1970)
14. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software: Practice and Experience **40**(12) (2010)
15. Khronos Compute Working Group: OpenACC Directives for Accelerators (2012). http://www.openacc-standard.org
16. Owens, J.: SuperComputing 07, High Performance Computing with CUDA tutorial (2007)